Contents

[**1.** **Depedency injection trong spring** 2](#_Toc180848774)

[\* Trình bày các annotation thường sử dụng? 2](#_Toc180848775)

[\* Khác nhau giữa component vs service? 2](#_Toc180848776)

[\* Trình bày các life-cycle và hoàn cảnh sử dụng? 2](#_Toc180848777)

[\* Spring lưu các object được khởi tạo tại đâu ? 3](#_Toc180848778)

[\* Quá trình khởi tạo object ? 3](#_Toc180848779)

[**2.** **Spring Configuration** 3](#_Toc180848780)

[ Sử dụng annotation @Configuration cùng @Bean: 3](#_Toc180848781)

[ Sử dụng .properties cùng Environment hoặc @Value 4](#_Toc180848782)

[**3. Spring RestfulAPI** 5](#_Toc180848783)

[**4. RestClient: Retrofit, RestTemplate,** 5](#_Toc180848784)

[**5. JPA và hibernate** 5](#_Toc180848785)

[- JPA, Hibernate là gì ? 5](#_Toc180848786)

[BT: So sánh sự khác biệt giữa các pattern ? (Điểm mạnh , yếu và tình huống sử dụng. Không cần chi tiết vào cách cài đặt) 6](#_Toc180848787)

[BT: Tạo sample project sử dụng JPA kết nối tới db. Project có đầy đủ method CRUD 7](#_Toc180848788)

[**6. Multiple threads and concurrency** 7](#_Toc180848789)

[- Khái niệm thread là gì? 7](#_Toc180848790)

[- Triển khai multiple thread trong java core: Trread, Callable,Runable,CompletableFuture,TaskExecutor,ForkJoinPool. 7](#_Toc180848791)

[ Thread 7](#_Toc180848792)

[• Runnable 8](#_Toc180848793)

[ CompletableFuture thực hiện async task 9](#_Toc180848794)

[ TaskExecutor 10](#_Toc180848795)

[ ForkJoinPool 11](#_Toc180848796)

[- ThreadSafe là gì ? Ví dụ về thread safe trong java 14](#_Toc180848797)

[- Concurrency? Phân biệt concurrency vs parallel. 14](#_Toc180848798)

[- Xử lý concurrency trong java 15](#_Toc180848799)

[ Thread 15](#_Toc180848800)

[ Executor Framework 15](#_Toc180848801)

[ Synchronization 15](#_Toc180848802)

[ Lock Interface 16](#_Toc180848803)

[ Concurrent Collections 16](#_Toc180848804)

[ Atomic Variables 16](#_Toc180848805)

1. **Depedency injection trong spring**

## \* Trình bày các annotation thường sử dụng?

* @Component: Đánh dấu class để Spring tự động phát hiện và quản lý bean.
* @Service: Giống @Component nhưng thể hiện rõ rằng class này cung cấp logic nghiệp vụ.
* @Repository: Chỉ định các class truy cập dữ liệu.
* @Autowired: Tiêm các dependency vào bean.
* @Bean: Định nghĩa một bean trong một class cấu hình

## \* Khác nhau giữa component vs service?

* @Service gắn cho các Bean đảm nhiệm xử lý logic, nghiệp vụ. Thường là các Bean trong tầng logic
* @Component gắn cho các Bean khác.

## \* Trình bày các life-cycle và hoàn cảnh sử dụng?

* Khi ApplicationContext tìm thấy một Bean cần quản lý, nó sẽ khởi tạo bằng Constructor
* inject dependencies vào Bean bằng Setter, và thực hiện các quá trình cài đặt khác vào Bean như setBeanName, setBeanClassLoader, v.v..
* Hàm đánh dấu @PostConstruct được gọi
* Bean sẵn sàng để hoạt động
* Nếu IoC Container không quản lý bean nữa hoặc bị shutdown nó sẽ gọi hàm @PreDestroy trong Bean
* Xóa Bean.

## \* Spring lưu các object được khởi tạo tại đâu ?

* Spring lưu trong ApplicationContext (IoC Container)

## \* Quá trình khởi tạo object ?

* Khi ApplicationContext phát hiện một class cần đối tượng bean, nó sẽ gọi constructor của class đó, sau đó tiêm các dependency, rồi gọi phương thức khởi tạo như @PostConstruct

# **2. Spring Configuration**

Sử dụng annotation @Configuration cùng @Bean; hoặc sử dụng file .properties cùng Environment hoặc @Value.

* Sử dụng annotation @Configuration cùng @Bean:
* @Configuration đánh dấu một class thuộc loại configuration
* @Bean nằm trong các class được đánh dấu @Configuration, đánh dấu các method để đưa vào Context
* Ví dụ

@Configuration

public class DatabaseConfig {

// MySQL DataSource

@Bean

public DataSource mysqlDataSource() {

return …

}

// PostgreSQL DataSource

@Bean

public DataSource postgresqlDataSource() {

return …

}

// MS SQL DataSource

@Bean

public DataSource mssqlDataSource() {

return …

}

}

* Sử dụng .properties cùng Environment hoặc @Value
* Cấu hình trong application.properties:

app.name=MySpringApp

* Dùng annotation @Value

@Value("${app.name}")

private String appName;

* Dùng Environment

@Autowired

private Environment env;

env.getProperty("app.name");

# **3. Spring RestfulAPI**

- https://spring.io/guides/tutorials/rest

- Nguyên tắc thiết kế api: https://learn.microsoft.com/en-us/azure/architecture/best-practices/api-design

BT: Tạo restful api: Đầy đủ các http method

* Link [my-internship/Part 3/Spring Boot example/src/main/java/com/backend/example at main · huuminhs/my-internship (github.com)](https://github.com/huuminhs/my-internship/tree/main/Part%203/Spring%20Boot%20example/src/main/java/com/backend/example)

# **4. RestClient: Retrofit, RestTemplate,**

Trong báo cáo sử dụng WebClient vì RestTemplate bị deprecated, không tìm được tài liệu

BT: Thử với Rest api ở BT-3 hoặc với api bất kỳ. (hai cách). Cấu hình API được khai báo ở application.properties (url,token ...)

* Link [my-internship/Part 3/WebClient.java at main · huuminhs/my-internship (github.com)](https://github.com/huuminhs/my-internship/blob/main/Part%203/WebClient.java)

# **5. JPA và hibernate**

## - JPA, Hibernate là gì ?

* Java Persistence API là một đặc tả java cho phép ánh xạ đối tượng java với bảng trong SQL.

Vì nó chỉ là đặc tả nên nó không làm gì cả.

* Hibernate là một triển khai (implement) của JPA

Một số data access pattern.

Repository pattern: https://www.geeksforgeeks.org/repository-design-pattern/

Specification pattern: https://en.wikipedia.org/wiki/Specification\_pattern

QueryObject pattern: https://martinfowler.com/eaaCatalog/queryObject.html

DataAccessObject

## BT: So sánh sự khác biệt giữa các pattern ? (Điểm mạnh , yếu và tình huống sử dụng. Không cần chi tiết vào cách cài đặt)

* **DAO (Data Access Object)**:
* **Ưu Điểm**: Tách biệt các truy vấn dữ liệu, giúp thay đổi nguồn dữ liệu mà không cần thay đổi mã khác.
* **Nhược Điểm**: Dễ dàng dẫn đến mã lặp và yêu cầu nhiều DAO cho nhiều đối tượng.
* **Trường Hợp Sử Dụng**: Thích hợp cho các ứng dụng phức tạp với nhiều nguồn dữ liệu.
* **Repository**:
* **Ưu Điểm**: Dễ sử dụng, tích hợp tốt với ORM, và thường có các phương thức CRUD sẵn có.
* **Nhược Điểm**: Có thể trở nên phức tạp khi logic truy vấn trở nên phức tạp.
* **Trường Hợp Sử Dụng**: Phù hợp cho ứng dụng CRUD, đặc biệt khi sử dụng các ORM như Hibernate.
* **Query Object**:
* **Ưu Điểm**: Tách biệt logic truy vấn, dễ dàng kiểm tra và sử dụng lại.
* **Nhược Điểm**: Tăng độ phức tạp của mã nếu không được tổ chức hợp lý.
* **Trường Hợp Sử Dụng**: Khi cần tái sử dụng logic truy vấn hoặc xây dựng các truy vấn phức tạp.
* **Specification**:
* **Ưu Điểm**: Hỗ trợ xây dựng truy vấn phức tạp và linh hoạt, dễ dàng kết hợp nhiều tiêu chí.
* **Nhược Điểm**: Cần phải quản lý nhiều đối tượng Specification, có thể gây khó khăn trong việc duy trì.
* **Trường Hợp Sử Dụng**: Khi cần định nghĩa các quy tắc phức tạp cho truy vấn và muốn có khả năng kết hợp linh hoạt các điều kiện.

## BT: Tạo sample project sử dụng JPA kết nối tới db. Project có đầy đủ method CRUD

* Link: [my-internship/Part 3/Spring Boot example/src/main/java/com/backend/example at main · huuminhs/my-internship (github.com)](https://github.com/huuminhs/my-internship/tree/main/Part%203/Spring%20Boot%20example/src/main/java/com/backend/example)

# **6. Multiple threads and concurrency**

## - Khái niệm thread là gì?

* Thread là một tập các câu lệnh thực thi, tạo thành đơn vị nhỏ nhất để giao cho bộ lên kế hoạch của hệ điều hành quản lý.

## - Triển khai multiple thread trong java core: Trread, Callable,Runable,CompletableFuture,TaskExecutor,ForkJoinPool.

* Thread

class MyThread extends Thread {

public void run() {

System.out.println("Thread " + Thread.currentThread().getId() + " is running");

}

public static void main(String[] args) {

for (int i = 0; i < 3; i++) {

MyThread thread = new MyThread();

thread.start(); // Khởi chạy thread mới

}

}

}

### • Runnable

class MyRunnable implements Runnable {

public void run() {

System.out.println("Thread " + Thread.currentThread().getId() + " is running");

}

public static void main(String[] args) {

for (int i = 0; i < 3; i++) {

Thread thread = new Thread(new MyRunnable());

thread.start();

}

}

}

* CompletableFuture thực hiện async task

import java.util.concurrent.CompletableFuture;

public class CompletableFutureExample {

public static void main(String[] args) {

// Tạo một CompletableFuture để thực hiện tác vụ bất đồng bộ

CompletableFuture<String> future = CompletableFuture.supplyAsync(() -> {

// Giả lập một tác vụ mất thời gian

try {

Thread.sleep(1000);

} catch (InterruptedException e) {

e.printStackTrace();

}

return "Kết quả từ tác vụ bất đồng bộ!";

});

// Đợi tác vụ hoàn thành và lấy kết quả

future.thenAccept(result -> System.out.println(result));

// Giữ main thread để xem kết quả

try {

Thread.sleep(2000);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

* TaskExecutor

import org.springframework.core.task.TaskExecutor;

@Component

public class TaskExecutorExample {

private final TaskExecutor taskExecutor;

public TaskExecutorExample(TaskExecutor taskExecutor) {

this.taskExecutor = taskExecutor;

}

public void executeTask() {

taskExecutor.execute(() -> {

System.out.println("Tác vụ được thực thi trong thread: " + Thread.currentThread().getName());

});

}

}

@Component

public class TaskRunner implements CommandLineRunner {

private final TaskExecutorExample taskExecutorExample;

public TaskRunner(TaskExecutorExample taskExecutorExample) {

this.taskExecutorExample = taskExecutorExample;

}

@Override

public void run(String... args) {

taskExecutorExample.executeTask();

}

}

* ForkJoinPool

import java.util.concurrent.RecursiveTask;

import java.util.concurrent.ForkJoinPool;

public class ForkJoinExample {

// Task tính tổng phần tử của một mảng

static class SumTask extends RecursiveTask<Long> {

private final int[] array;

private final int start, end;

private static final int THRESHOLD = 10; // Ngưỡng chia nhỏ

public SumTask(int[] array, int start, int end) {

this.array = array;

this.start = start;

this.end = end;

}

@Override

protected Long compute() {

if (end - start <= THRESHOLD) {

// Nếu số lượng phần tử nhỏ hơn ngưỡng, tính trực tiếp

long sum = 0;

for (int i = start; i < end; i++) {

sum += array[i];

}

return sum;

} else {

// Ngược lại, chia thành hai task nhỏ hơn

int mid = (start + end) / 2;

SumTask leftTask = new SumTask(array, start, mid);

SumTask rightTask = new SumTask(array, mid, end);

// Chạy hai task song song

leftTask.fork();

long rightResult = rightTask.compute();

long leftResult = leftTask.join();

// Kết hợp kết quả

return leftResult + rightResult;

}

}

}

public static void main(String[] args) {

int[] array = new int[100];

for (int i = 0; i < array.length; i++) {

array[i] = i + 1;

}

ForkJoinPool pool = new ForkJoinPool();

SumTask task = new SumTask(array, 0, array.length);

long result = pool.invoke(task);

System.out.println("Tổng các phần tử trong mảng là: " + result);

}

}

## - ThreadSafe là gì ? Ví dụ về thread safe trong java

* Trong java có tình huống khi nhiều luồng cùng chạy, chúng cùng đọc ghi một tài nguyên găng. Điều đó gây ra kết quả khác nhau với mỗi lần chạy khác nhau.

Thread-safe là khi chạy đa luồng và không xuất hiện tình huống trên.

* Ví dụ: ConcurrentHashMap
  + Giống HashMap bình thường nhưng chia thành các đoạn nhỏ, khi một thread đọc ghi trên một đoạn thì chỉ lock đoạn đó
  + ConcurrentHashMap<String, Integer> map = new ConcurrentHashMap<>(32); // Sử dụng 32 đoạn

## - Concurrency? Phân biệt concurrency vs parallel.

* Concurrency là xử lý nhiều task trong một khoảng thời gian, nhưng không đồng thời. Tức là xử lý task 1 một lúc rồi chuyển sang task 2, rồi tiếp tục với task 3,...
* Parallel là xử lý song song, tức là task 1, task 2, task 3,.. chạy song song với nhau

## - Xử lý concurrency trong java

* Thread

Java cho phép tạo và quản lý các luồng (thread) để thực hiện các tác vụ đồng thời. Bạn có thể tạo một lớp kế thừa từ Thread hoặc cài đặt giao diện Runnable.

class MyThread extends Thread {

public void run() {

// Thực hiện tác vụ

}

}

MyThread thread = new MyThread();

thread.start();

* Executor Framework

Cung cấp một cách dễ dàng hơn để quản lý và sử dụng các luồng thông qua lớp ExecutorService

ExecutorService executor = Executors.newFixedThreadPool(10);

executor.submit(() -> {

// Thực hiện tác vụ

});

executor.shutdown();

* Synchronization

Để đảm bảo rằng một đoạn mã chỉ được thực thi bởi một luồng tại một thời điểm nhất định, bạn có thể sử dụng từ khóa synchronized.

public synchronized void synchronizedMethod() {

// Đoạn mã được đồng bộ hóa

}

* Lock Interface

Cung cấp khả năng kiểm soát tốt hơn so với synchronized. Bạn có thể sử dụng ReentrantLock để khóa và mở khóa tài nguyên.

Lock lock = new ReentrantLock();

lock.lock();

try {

// Thực hiện tác vụ

} finally {

lock.unlock();

}

* Concurrent Collections

Java cung cấp một số collection đồng thời như ConcurrentHashMap, CopyOnWriteArrayList giúp bạn xử lý dữ liệu an toàn khi nhiều luồng cùng truy cập.

* Atomic Variables

Sử dụng các lớp như AtomicInteger, AtomicBoolean để thao tác an toàn với các biến nguyên thủy.

AtomicInteger count = new AtomicInteger(0);

count.incrementAndGet();

Tài liệu:

- https://www.geeksforgeeks.org/multithreading-in-java/

- https://www.baeldung.com/java-completablefuture

- https://www.geeksforgeeks.org/difference-between-concurrency-and-parallelism/